Суперкомпьютеры и параллельная обработка данных

Бахтин Владимир Александрович
к.ф.-м.н., ведущий научный сотрудник
Института прикладной математики им М.В.Келдыша РАН
кафедра системного программирования
факультет вычислительной математики и кибернетики
Московского университета им. М.В. Ломоносова

Понятие задачи. Директива task

Явные задачи (explicit tasks) задаются при помощи директивы:
#pragma omp task [клауза[[,] клауза] ...]
структурный блок
где клауза одна из :

- **if (scalar-expression)**
- **final(scalar-expression)** **//OpenMP 3.1**
- **untied**
- **mergeable** **//OpenMP 3.1**
- **shared (list)**
- **private (list)**
- **firstprivate (list)**
- **default (shared | none)**
- **depend (dependence-type: list)** **//OpenMP 4.0**

В результате выполнения директивы task создается новая задача, которая состоит из операторов структурного блока; все используемые в операторах переменные могут быть локализованы внутри задачи при помощи соответствующих клауз. Созданная задача будет выполнена одной нитью из группы.

Понятие задачи. Директива task

```
#pragma omp for schedule(dynamic)
    for (i=0; i<n; i++) {
        func(i);
    }
```

```
#pragma omp single
{
    for (i=0; i<n; i++) {
        #pragma omp task firstprivate(i)
            func(i);
    }
}
```

Использование директивы task

```
typedef struct node node;
struct node {
    int data;
    node * next;
};
void increment_list_items(node * head)
{
   #pragma omp parallel
      {
        #pragma omp single
          {
            node * p = head;
            while (p) {
              #pragma omp task
                  process(p);
              p = p->next;
            }
          }
      }
}
```

Использование директивы task. Клауза if

```
double *item;
int main() {
  #pragma omp parallel shared (item)
    {
      #pragma omp single
        {
          int size;
          scanf("%d",&size);
          item = (double*)malloc(sizeof(double)*size);
          for (int i=0; i<size; i++)
             #pragma omp task if (size > 10)
                 process(item[i]);
        }
    }
}
```

Если накладные расходы на организацию задач превосходят время, необходимое для выполнения блока операторов этой задачи, то блок операторов будет немедленно выполнен нитью, выполнившей директиву task

Использование директивы task

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
  #pragma omp parallel shared (item)
    {
      #pragma omp single
        {
          for (int i=0; i<LARGE_NUMBER; i++)
            #pragma omp task
              process(item[i]);
        }
    }
}
```

Как правило, в компиляторах существуют ограничения на количество создаваемых задач. Выполнение цикла, в котором создаются задачи, будет приостановлено. Нить, выполнявшая этот цикл, будет использована для выполнения одной из задач

Использование директивы task. Клауза untied

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
    #pragma omp parallel
      {
        #pragma omp single
          {
              #pragma omp task untied
             {
                for (int i=0; i<LARGE_NUMBER; i++)
                   #pragma omp task
                      process(item[i]);
             }
          }
      }
}
```

Клауза untied - выполнение задачи после приостановки может быть продолжено любой нитью группы

Использование задач. Директива taskwait

```
#pragma omp taskwait

int fibonacci(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
            i=fibonacci (n-1);
        #pragma omp task shared(j)
            j=fibonacci (n-2);
       #pragma omp taskwait
        return i+j;
    }
}
```

```
int main () {
    int res;
    #pragma omp parallel
    {
        #pragma omp single
        {
            int n;
            scanf("%d",&n);
            #pragma omp task shared(res)
                res = fibonacci(n);
        }
    }
    printf ("Finonacci number =  %d\n", res);
}
```

Использование директивы task. Клауза final

```
int fib (int n, int d) {
    int x, y;
    if (n < 2) return 1;
#pragma omp task final (d > LIMIT) mergeable
        x =  fib (n - 1, d + 1);
#pragma omp task final (d > LIMIT) mergeable
        y =  fib (n - 2, d + 1);
#pragma omp taskwait
        return x + y;
}

int omp_in_final (void);
```

Зависимости между задачами (OpenMP 4.0)

Клауза depend(dependence-type : list)
где dependence-type:
- **in**
- **out**
- **inout**

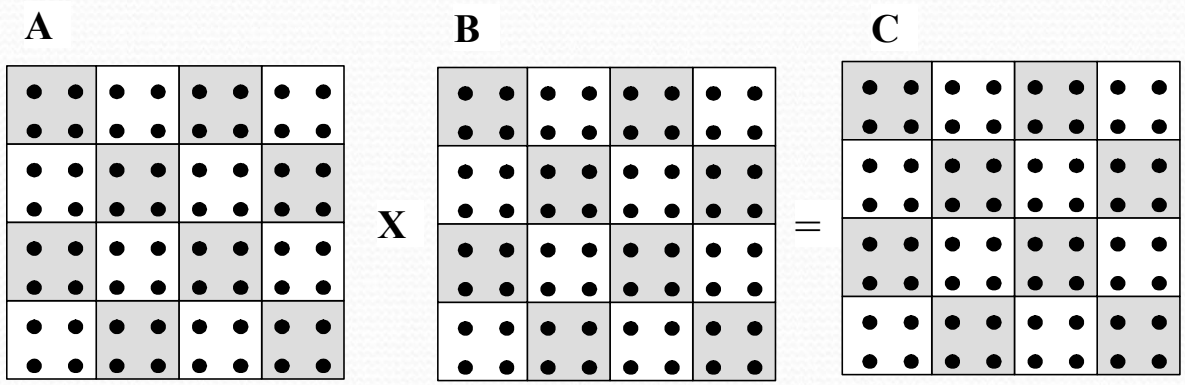
```
int i, y, a[100];

#pragma omp task depend(out : a)
{
   for (i=0;i<100; i++) a[i] = i + 1;
}

#pragma omp task depend(in : a[0:50]) depend(out : y)
{
   y = 0;
   for (i=0;i<50; i++) y += a[i];
}

#pragma omp task depend(in : y) {
   printf("%d\n", y);
}
```

Умножение матриц



$$\begin{pmatrix} A_{00}A_{01}\ldots A_{0q-1} \\ \ldots \\ A_{q-10}A_{q-11}\ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01}\ldots B_{0q-1} \\ \ldots \\ B_{q-10}B_{q-11}\ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01}\ldots C_{0q-1} \\ \ldots \\ c_{q-10}C_{q-11}\ldots C_{q-1q-1} \end{pmatrix}, \quad C_{ij} = \sum_{s=1}^{q} A_{is}B_{sj}$$

Зависимости между задачами (OpenMP 4.0)

```
void matmul_depend (int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
     #pragma omp task private(ii, jj, kk) firstprivate(i, j, k) \
         depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
         depend ( inout: C[i:BS][j:BS] )
        for (ii = i; ii < i+BS; ii++ )
         for (jj = j; jj < j+BS; jj++ )
           for (kk = k; kk < k+BS; kk++ )
             C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

Приоритет задачи (OpenMP 4.5)

```
void compute_array (float *node, int M);

void compute_matrix (float *array, int N, int M)
{
 int i;
 #pragma omp parallel private(i)
 #pragma omp single
 {
    for (i=0;i<N; i++) {
      #pragma omp task priority(i)
          compute_array(&array[i*M], M);
    }
 }
}
```

Task Affinity (OpenMP 5.0)

```
double * alloc_init_B(double *A, int N);
void compute_on_B(double *B, int N);

void task_affinity(double *A, int N)
{
  double * B;
  #pragma omp task depend(out:B) shared(B) affinity(A[0:N])
  {
    B = alloc_init_B(A,N);
  }
  #pragma omp task depend( in:B) shared(B) affinity(A[0:N])
  {
    compute_on_B(B,N);
  }
  #pragma omp taskwait
}
```

Директива taskloop (OpenMP 4.5)

#pragma omp taskloop [clause[[,]clause]...]
structured-block

где ***clause*** *– одна из:*

if([taskloop :]scalar-expr)
shared(list)
private(list)
firstprivate(list)
lastprivate(list)
default(shared | none)
grainsize(grain-size)
num_tasks(num-tasks)
collapse(n)
final(scalar-expr)
priority(priority-value)
untied
mergeable
nogroup

Директива taskloop (OpenMP 4.5)

```
for (i = 0; i<SIZE; i++) {
        A[i]=A[i]*B[i]*S;
}

for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}

for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE : i+TS;
    #pragma omp task private(ii) \
          firstprivate(i,UB) shared(S,A,B)
    for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
    }
}
```

Директива taskloop (OpenMP 4.5)

```
for (i = 0; i<SIZE; i++) {
        A[i]=A[i]*B[i]*S;
}

for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}

for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE : i+TS;
    #pragma omp task private(ii) \
          firstprivate(i,UB) shared(S,A,B)
    for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
    }
}
```

```
#pragma omp taskloop grainsize(TS)
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

Содержание

Конструкции для синхронизации нитей

- Директива master
- Директива critical
- Директива atomic
- Семафоры
- Директива barrier
- Директива taskyield
- Директива taskwait
- Директива taskgroup // OpenMP 4.0

Директива master

```
#pragma omp master
структурный блок
/*Структурный блок будет выполнен MASTER-нитью группы. По завершении выполнения структурного блока барьерная синхронизация нитей не выполняется*/

#include <stdio.h>
void init(float *a, float *b ) {
  #pragma omp master
      scanf("%f %f", a, b);
  #pragma omp barrier
}
int main () {
   float x,y;
   #pragma omp parallel
   {
      init (&x,&y);
      parallel_work (x,y);
   }
}
```

Вычисление числа π на OpenMP с использованием критической секции

```
#include <omp.h>
int main ()
{
   int n =100000, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
   {
     double local_sum = 0.0;
 #pragma omp for nowait
      for (i = 1; i <= n;  i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
       }
 #pragma omp critical
            sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f”, pi);
    return 0;
}
```

Использование критической секции

```
int *next_from_queue(int type);
void work(int *val);

void critical_example()
{
  #pragma omp parallel
  {
    int *ix_next, *iy_next;
    #pragma omp critical (xaxis)
         ix_next = next_from_queue(0);
    work(ix_next);

    #pragma omp critical (yaxis)
        iy_next = next_from_queue(1);
    work(iy_next);
   }
}
```

Директива atomic

```
#pragma omp atomic [ read | write | update | capture ] [seq_cst]
      expression-stmt

#pragma omp atomic capture
      structured-block
```

Если указана клауза read:

```
      v = x;
```

Если указана клауза write:

```
      x = expr;
```

Если указана клауза update или клаузы нет, то expression-stmt:

```
x binop= expr;
x = x binop expr;
x++;
++x;
x--;
--x;
```

x – скалярная переменная, expr – выражение, в котором не присутствует переменная x.

binop - не перегруженный оператор:

+ , * , - , / , & , ^ , | , << , >>

binop=:

++ , --

Директива atomic

Если указана клауза capture, то expression-stmt:

```
v = x++;
v = x--;
v = ++x;
v = -- x;
v = x binop= expr;
```

Если указана клауза capture, то structured-block:

```
{ v = x; x binop= expr;}
{ v = x; x =  x binop expr;}
{ v = x; x++;}
{ v = x; ++x;}
{ v = x; x--;}
{ v = x; --x;}
{ x binop= expr; v = x;}
{ x =  x binop expr;  v = x;}
{ v = x; x binop= expr;}
{ x++; v = x;}
{ ++ x ; v = x;}
{ x--; v = x;}
{ --x; v = x;}
```

Встроенные функции для атомарного доступа к памяти в GCC

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)
type __sync_add_and_fetch (type *ptr, type value, ...)
type __sync_sub_and_fetch (type *ptr, type value, ...)
type __sync_or_and_fetch (type *ptr, type value, ...)
type __sync_and_and_fetch (type *ptr, type value, ...)
type __sync_xor_and_fetch (type *ptr, type value, ...)
type __sync_nand_and_fetch (type *ptr, type value, ...)
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
```

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

Вычисление числа π на OpenMP с использованием директивы atomic

```
int main ()
{
   int n =100000, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
   {
     double local_sum = 0.0;
#pragma omp for nowait
      for (i = 1; i <= n;  i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
       }
#pragma omp atomic
            sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Использование директивы atomic

```
int atomic_read(const int *p)
{
   int value;
   /* Guarantee that the entire value of *p is read atomically. No part of
   * *p can change during the read operation.
   */
   #pragma omp atomic read
   value = *p;
   return value;
}

void atomic_write(int *p, int value)
{
   /* Guarantee that value is stored atomically into *p. No part of *p can change
   * until after the entire write operation is completed.
   */
   #pragma omp atomic write
   *p = value;
}
```

Использование директивы atomic

```
int fetch_and_add(int *p)
{
  /* Atomically read the value of *p and then increment it. The previous value is
  * returned. */
  int old;
  #pragma omp atomic capture
  { old = *p; (*p)++; }
  return old;
}
```

seq_cst - sequentially consistent atomic construct, the operation to have the same meaning as a memory_order_seq_cst atomic operation in C++11/C11

```
#pragma omp atomic capture seq_cst  // OpenMP 4.0
{--x; v = x;} // capture final value of x in v and flush all variables
```

Семафоры

Концепцию семафоров описал Дейкстра (Dijkstra) в 1965

Семафор - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:

P - функция запроса семафора
P(s): [if (s == 0) <заблокировать текущий процесс>; else s = s-1;]

V - функция освобождения семафора
V(s): [if (s == 0) <разблокировать один из заблокированных процессов>; s = s+1;]

Семафоры в OpenMP

Состояния семафора:

- ❑uninitialized
- ❑unlocked
- ❑locked

```
void omp_init_lock(omp_lock_t *lock); /* uninitialized to unlocked*/
void omp_destroy_lock(omp_lock_t *lock); /* unlocked  to uninitialized */
void omp_set_lock(omp_lock_t *lock);  /*P(lock)*/
void omp_unset_lock(omp_lock_t *lock); /*V(lock)*/
int omp_test_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Вычисление числа π с использованием семафоров

```
int main ()
{
   int n =100000, i; double pi, h, sum, x;
   omp_lock_t lck;
   h   = 1.0 / (double) n;
   sum = 0.0;
   omp_init_lock(&lck);
#pragma omp parallel default (none) private (i,x) shared (n,h,sum,lck)
   {
      double local_sum = 0.0;
      #pragma omp for nowait
      for (i = 1; i <= n;  i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
      }
      omp_set_lock(&lck);
           sum += local_sum;
      omp_unset_lock(&lck);
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    omp_destroy_lock(&lck);
    return 0;
}
```

Использование семафоров

```
#include <stdio.h>
#include <omp.h>
int main()
{
   omp_lock_t lck;
   int id;
   omp_init_lock(&lck);
   #pragma omp parallel shared(lck) private(id)
   {
      id = omp_get_thread_num();
      omp_set_lock(&lck);
      printf("My thread id is %d.\n", id); /* only one thread at a time can execute this printf */
      omp_unset_lock(&lck);
      while (! omp_test_lock(&lck)) {
            skip(id); /* we do not yet have the lock, so we must do something else*/
      }
      work(id); /* we now have the lock and can do the work */
      omp_unset_lock(&lck);
   }
   omp_destroy_lock(&lck);
   return 0;
}
```

```
void skip(int i) {}
void work(int i) {}
```

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_lock_t lck;
} pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_lock(&p->lck);
    p->b += b;
    omp_unset_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
   #pragma omp parallel sections
   {
      #pragma omp section
         incr_pair(p,1,2);
      #pragma omp section
         incr_b(p,3);
    }
}
```

Deadlock!

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_nest_lock_t lck;
} pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
   #pragma omp parallel sections
   {
      #pragma omp section
         incr_pair(p,1,2);
      #pragma omp section
         incr_b(p,3);
    }
}
```

Директива barrier

Точка в программе, достижимая всеми нитями группы, в которой выполнение программы приостанавливается до тех пор пока все нити группы не достигнут данной точки и все задачи, выполняемые группой нитей будут завершены.

#pragma omp barrier

По умолчанию барьерная синхронизация нитей выполняется:

- по завершению конструкции **parallel**;
- при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**) , если не указана клауза **nowait**.

```
#pragma omp parallel
{
   #pragma omp master
   {
      int i, size;
      scanf("%d",&size);
      for (i=0; i<size; i++) {
         #pragma omp task
            process(i);
      }
   }
   #pragma omp barrier
}
```

Директива taskyield

```
#include <omp.h>
void something_useful ( void );
void something_critical ( void );
void foo ( omp_lock_t * lock, int n )
{
    int i;
    for ( i = 0; i < n; i++ )
        #pragma omp task
        {
            something_useful();
            while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

Директива taskwait

```
#pragma omp taskwait

int fibonacci(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
            i=fibonacci (n-1);
        #pragma omp task shared(j)
            j=fibonacci (n-2);
       #pragma omp taskwait
        return i+j;
    }
}
```

```
int main () {
    int res;
    #pragma omp parallel
    {
        #pragma omp single
        {
            int n;
            scanf("%d",&n);
            #pragma omp task shared(res)
                res = fibonacci(n);
        }
    }
    printf ("Finonacci number =  %d\n", res);
}
```

Директива taskwait

```
#pragma omp task {}  // Task1
#pragma omp task // Task2
{
   #pragma omp task {} // Task3
}
#pragma omp task {} // Task4

#pragma omp taskwait
// Гарантируется что в данной точке завершатся Task1 && Task2  && Task4
```

Директива taskgroup

```
#pragma omp task {}  // Task1
#pragma omp taskgroup
{
  #pragma omp task // Task2
  {
    #pragma omp task {} // Task3
  }
  #pragma omp task {} // Task4
}
// Гарантируется что в данной точке завершатся Task2 && Task3  && Task4
```